Context: Plan-Space Planning

In **state-space planning**, a program searches through a space of *world states*, seeking to find a path or paths that will take it from its initial state to a goal state.

State-space planning is too inflexible, because:

- it creates plans that are total orderings of a set of steps, and
- it assembles these plans in exactly the same order.
Plan-Space Planning Redux

In plan-space planning, a program searches through a space of plans, seeking a plan that will take it from its initial state to a goal state.

In this approach, we redefine some of the terms of our search:

- A **plan** is a set of steps and a set of constraints on the ordering of the steps.
- A **state** is a plan.
- The **goal state** is a plan that achieves all specified goals.
- An **operator** creates a new plan from an old plan.
Kinds of Operators

A refinement operator

- takes as input a partial plan, and
- adds either a step or a constraint to it.

That is, it makes the plan *more specific* by making one or more decisions left open in the partial plan.

A modification operator

- changes a constraint, or
- removes a step or a constraint, or
- does some combination of the two.

Whereas refinement operators allow the planner to “move forward” toward a goal, modification operators allow the planner to *back up*. 
What is a Plan?

A plan, whether partial or complete, consists of:

- a specification of its precondition state and its postcondition state
- a set of actions, or “steps”, $S_i$
- a set of orderings on steps, \{ $(S_i < S_j)$, $\ldots$ $\}$

An example of a partial plan:

- **Precondition**
  
  \[
  \text{armEmpty and clear( A ) and on( A, B ) and on( B, TABLE )}
  \]

- **Postcondition**
  
  \[
  \text{armEmpty and clear( B ) and on( B, A ) and on( A, TABLE )}
  \]

- **$S = \{ S_1, S_2 \}$**
  
  $S_1 = \text{stack( B, A )}$
  
  $S_2 = \text{stack( A, TABLE )}$

- **ORDER = \{ $(S_2 < S_1)$, $\ldots$ $\}$**
How Do We Make Plans?

A plan-space planning algorithm will do something like:

1. \( P := \text{empty-plan}(I, G) \)

2. Loop:
   a. If \( P \) is a solution, return \( P \).
   b. Choose \( F := \text{find-flaw}(P) \)
   c. Choose \( M := \text{find-method}(P, F) \)
   d. If there is no such method, return failure.
   e. \( P := \text{fix-flaw}(P, F, M) \)

This algorithm introduces some new concepts...

- An empty plan is a plan with no steps and no constraints.

  This plan says, “Yeah, I plan to get from A to B,” but does not contain actions to do it.

- A solution is any plan that achieves the \( I \rightarrow G \).

  So, Step2a is where we do our goal test in this algorithm.
Flaws and Methods

1. \( P := \text{empty-plan}(I, G) \)

2. Loop:
   a. If \( P \) is a solution, return \( P \).
   b. Choose \( F := \text{find-flaw}(P) \)
   c. Choose \( M := \text{find-method}(P, F) \)
   d. If there is no such method, return failure.
   e. \( P := \text{fix-flaw}(P, F, M) \)

A flaw is anything wrong with a plan.

- It might be something that is undone, such as “no action achieves this part of the goal” or “no action achieves this precondition of a step in the plan”.
- However, this algorithm can construct a partial plan that is internally inconsistent. (How?)

In such a case, a flaw can be an inconsistency, such as executing one step might undo a precondition for another step.

A method is a way to fix a flaw.

Usually, a flaw is a something undone, and so a method might add a step or a constraint to the plan.
A Demo of Plan-Space Planning

Assume that a robot is given this set of operators:

\[
\text{stack}(x, y)
\]

precondition: clear(y), holding(x)
add: \text{armEmpty}, on(x, y)
delete: clear(y), holding(x)

\[
\text{unstack}(x, y)
\]

precondition: on(x, y), clear(x), \text{armEmpty}
add: holding(x), clear(y)
delete: on(x, y), \text{armEmpty}

\[
\text{pickup}(x)
\]

precondition: clear(x), on(x, \text{TABLE}), \text{armEmpty}
add: holding(x)
delete: on(x, \text{TABLE}), \text{armEmpty}

\[
\text{putdown}(x)
\]

precondition: holding(x)
add: on(x, \text{TABLE}), \text{armEmpty}
delete: holding(x)

Solve:

Initial state: B A

Goal state: A B
demonstration of above
An Exercise

Assume that a robot is given this set of operators:

stack( x, y )

precondition: clear( y ), holding( x )
add: armEmpty, on( x, y )
delete: clear( y ), holding( x )

unstack( x, y )

precondition: on( x, y ), clear( x ), armEmpty
add: holding( x ), clear( y )
delete: on( x, y ), armEmpty

pickup( x )

precondition: clear( x ), on( x, TABLE ), armEmpty
add: holding( x )
delete: on( x, TABLE ), armEmpty

putdown( x )

precondition: holding( x )
add: on( x, TABLE ), armEmpty
delete: holding( x )

Solve:

Initial state:  

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Goal state:  

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
solution to above
Another Exercise

stack( x, y )

precondition: clear( y ), holding( x )
add: armEmpty, on( x, y )
delete: clear( y ), holding( x )

unstack( x, y )

precondition: on( x, y ), clear( x ), armEmpty
add: holding( x ), clear( y )
delete: on( x, y ), armEmpty

pickup( x )

precondition: clear( x ), on( x, TABLE ), armEmpty
add: holding( x )
delete: on( x, TABLE ), armEmpty

putdown( x )

precondition: holding( x )
add: on( x, TABLE ), armEmpty
delete: holding( x )

Solve:

Initial state: | Goal state:
------- | ------
A | A
B | B
C

C
A
B
C
solution to above
Flaws and Fixes in a Program

How can a program uses this approach to make plans?

The interesting new ideas here are:

- What is a flaw in a plan?
- What is a method for fixing a flaw?
- How does the program identify each?

First, a formal definition:

A proposition \( a \) is **necessarily true** before executing step \( s \) in plan \( p \) if both of the following are true:

- There is a step \( s_p \) in \( p \) such that \( s_p \) necessarily comes before \( s \) and \( s_p \) adds \( a \).

- For every step \( s_d \) in \( p \) that may delete \( a \), either \( s_d \) necessarily comes before \( s_p \) or \( s_d \) necessarily comes after \( s \).

What does “necessarily” mean here?
Using the Modal Truth Criterion

Now, we can define flaws and methods:

- A flaw is any precondition $a$ of a step $s$ that is not necessarily true before executing $s$.

- To fix a flaw, do both of the following:
  - Make sure that $a$ is made true before executing $s$.
    You can add a new step $s_p$ and make it necessarily prior to $s$.
    Or you can choose an $s_p$ that is already in the plan and add an ordering constraint.
  - Make sure that $a$ is not clobbered by some $s_d$.
    You can change the variable bindings on some $s_d$ so that it necessarily does not delete $a$.
    Or you can add an ordering so that $s_d$ must either come before $s_p$ or come after $s$. 
Applying the MTC

A flaw is any precondition $a$ of a step $s$ that is not necessarily true before executing $s$.

To fix a flaw, do both of the following:

- Make sure that $a$ is made true before executing $s$.
- Make sure that $a$ is not clobbered by some $s_d$.

Example 1:

- put on left sock
- put on left shoe
- put on right shoe

Example 2:

- put on left sock
- put on left shoe
- take off left sock
Partial-Order Planning

This style of planning is called *partial-order planning* (POP), because it enables a planner to construct plans that are only partially ordered and thus only complete enough to accomplish its goal.

Such a plan leaves the agent that will use it as much flexibility as possible at “execution time”.

The POP algorithm that uses the MTC and causal links is the culmination of a progression of increasingly more sophisticated planning algorithms.

POP satisfies our three key ideas from two sessions ago:

- States and operators are decomposable.
- It can add an action to the plan at any place.
- It decomposes a problem into sub-tasks, solves them separately, and re-assemble the solutions.

Sometimes, though, it comes up short in practice. So it is the subject of continued research!